**暨南大学本科实验报告专用纸**

课程名称 操作系统原理实验 成绩评定

实验项目名称 地址映射与内存共享 指导教师 郝振明

实验项目编号 0806015705 实验项目类型 验证 实验地点

学生姓名 张印祺 学号 2018051948

学院 信息科学技术学院 系 计算机科学系 专业 网络工程

实验时间 2020 年 5 月 17日 上 午～ 5 月 17 日 上 午

温度 ℃湿度

1. **实验目的**

深入理解操作系统的段、页式内存管理。包括理解段表、页表，以及逻辑地址、线性地址、物理地址的映射过程。

查看二级页表映射信息，理解页目录和页表的管理方式。

编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

**二、 预备知识**

请读者阅读《Linu内核完全注释》第4章的前4节，第5.3节以及第13章的内容。

**三、 实验内容**

3.1 准备实验

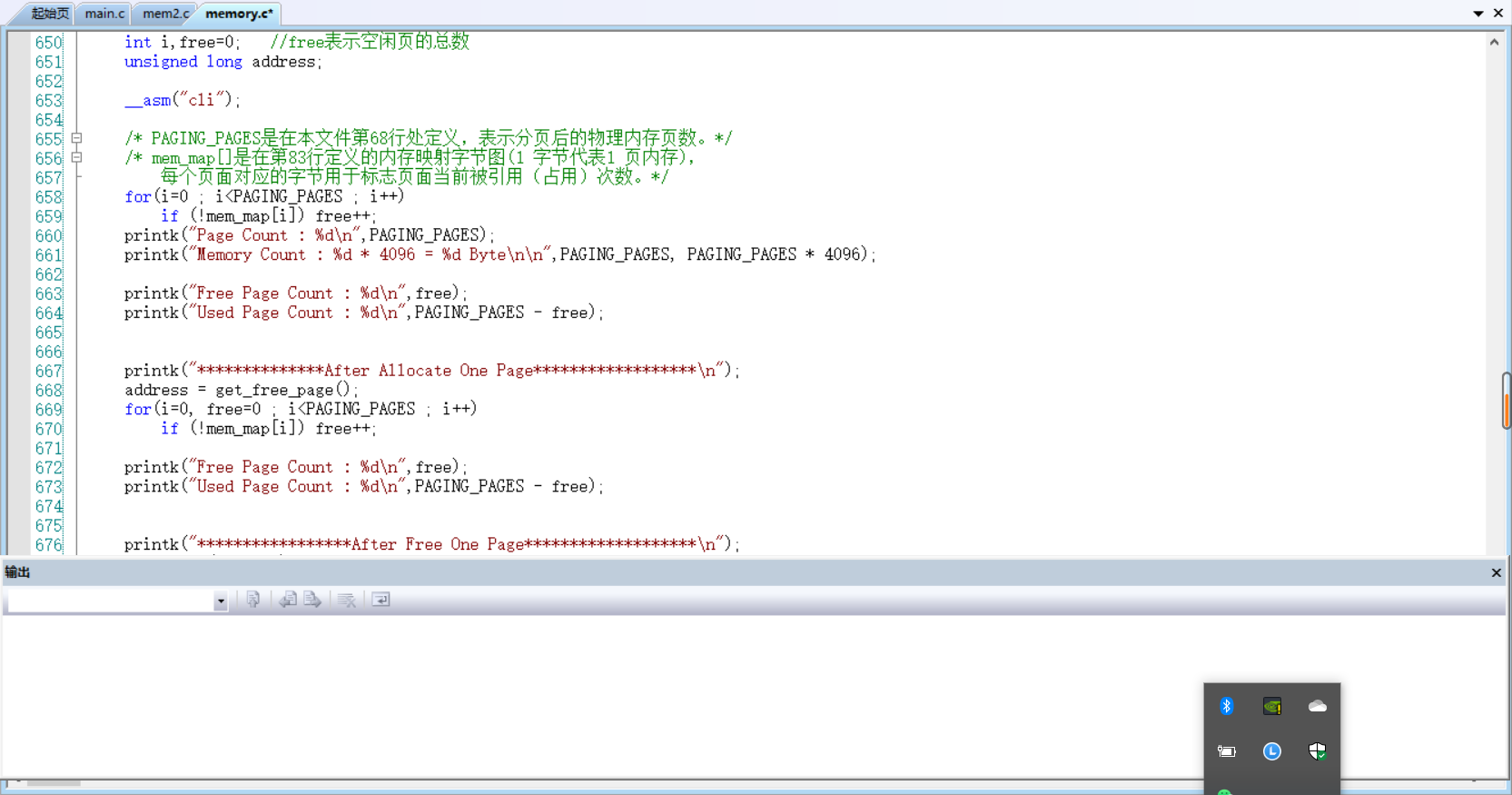
1. 启动 Engintime Linux Lab。

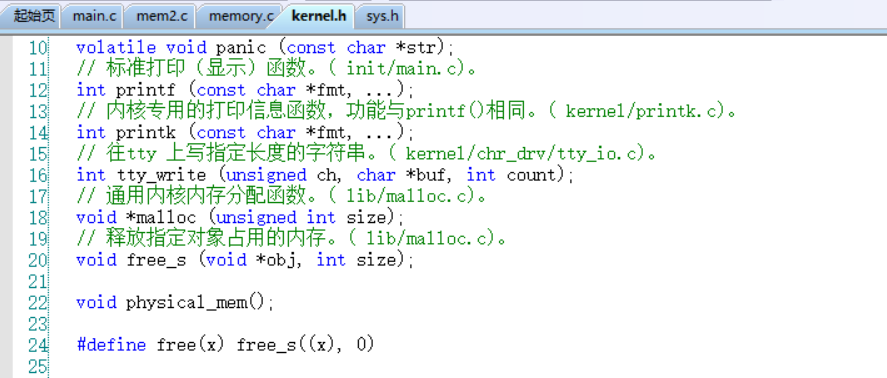
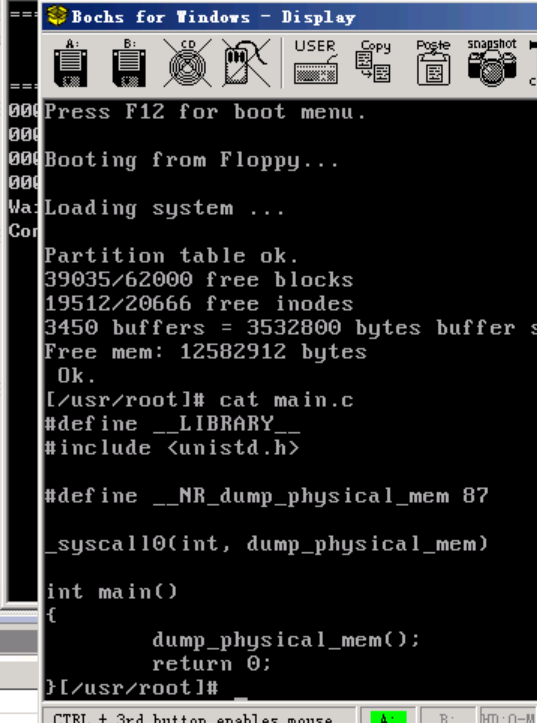
2. 新建一个 Linux011 Kernel 项目。

3.2 查看物理存储器的信息 请读者按照下面的步骤在Linux011 Kernel项目中添加一个系统调用函数，该函数用来 在终端设备上显示物理存储器的信息，包括物理页总数、空闲页数量和占用页数量。

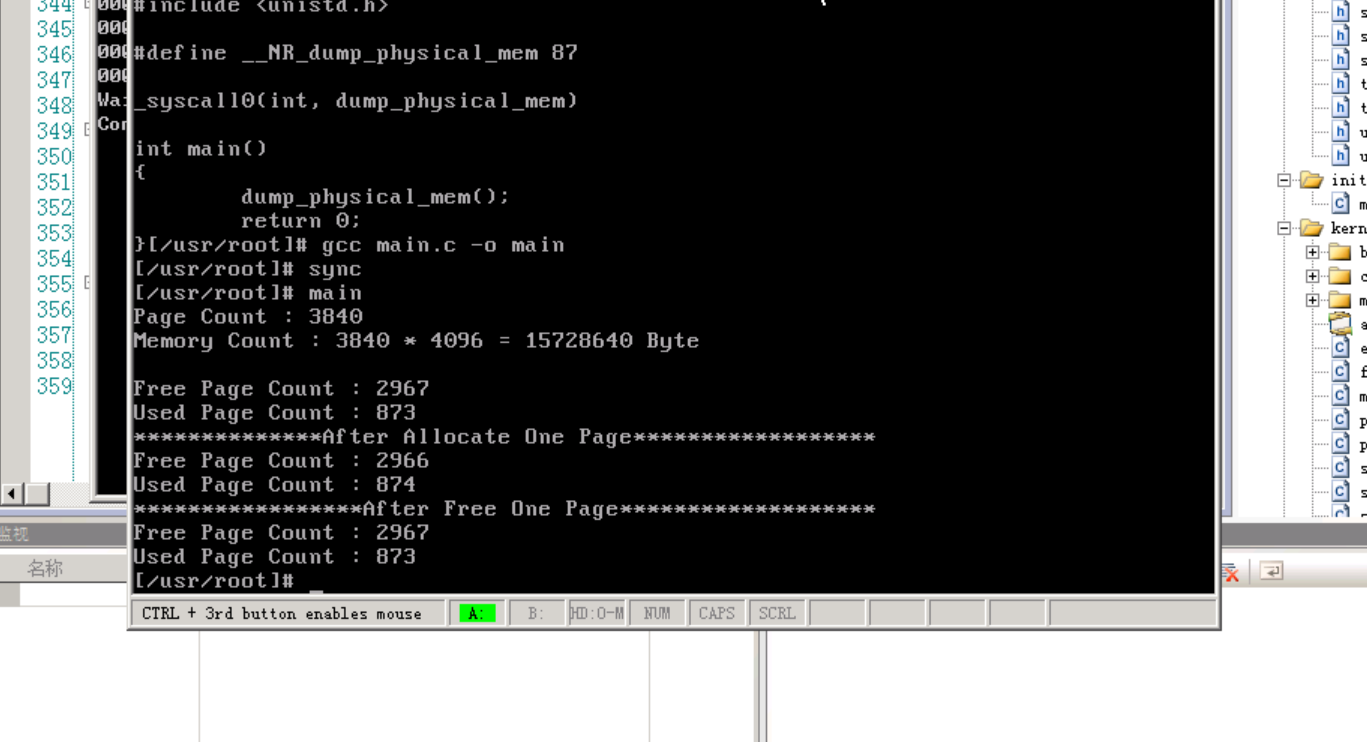
1. 打开“学生包”，在本实验对应文件夹下找到mem.c文件，拖动到Linux Lab中释放， 即可打开此文件。

将其中的函数physical\_mem复制到Linux011 Kernel项目下的 mm/memory.c 文件中的末尾

处，并且需在include/linux/kernel.h中添加函数的声 明。physical\_mem函数中的代码比较简单，请读者结合其中的注释自行理解。

2. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四)，该系 统调用的内核函数int dump\_physical\_mem可以写在 kernel/sys.c 文件的末尾， 在此函数中直接调用mm/memory.c文件中的physical\_mem函数即可。 3. 按F7生成项目，确保没有语法错误和警告。

4. 按F5启动调试，待Linux 0.11完全启动后，使用vi编辑器新建一个main.c文件，其 源代码如下所示： #define \_\_LIBRARY\_\_ #include <unistd.h> #define \_\_NR\_dump\_physical\_mem 87 \_syscall0(int, dump\_physical\_mem) int main() { dump\_physical\_mem(); return 0; }

5. 保存main.c文件并退出vi编辑器后，依次执行如下命令： 

Linux 内核实验教程

接下来对 physical\_mem 函数中的源代码进行修改，查看分配一个物理页和回收一个物 理页后物理存储器的变化。

1. 结束之前的调试。

2. 打开“学生包”，在本实验对应文件夹下找到 mem2.c 文件，拖动到 Linux Lab 中释 放，即可打开此文件。使用其中的函数physical\_mem替换之前添加的physical\_mem 函数。

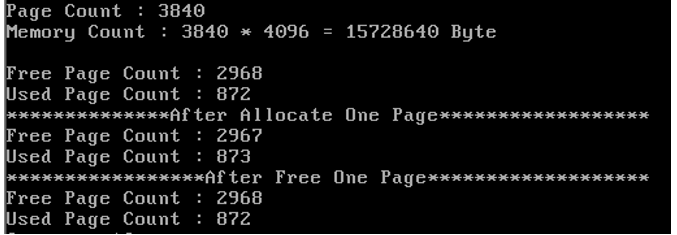
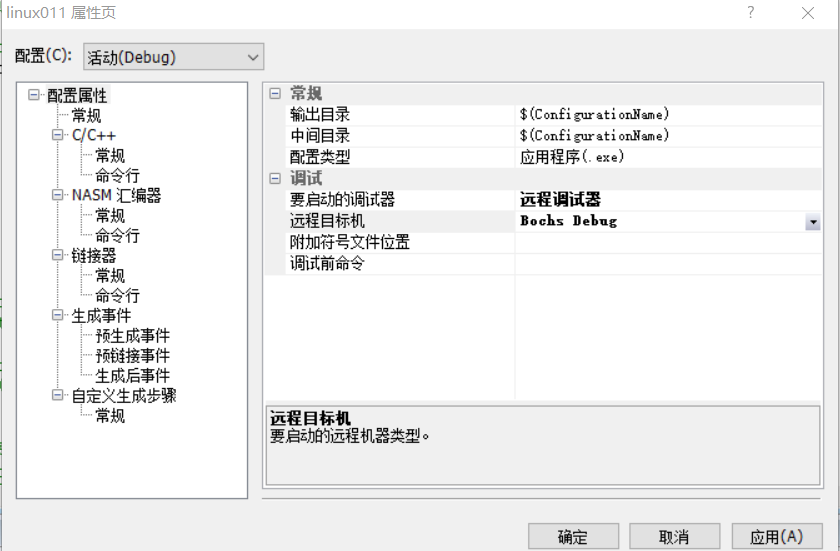
3. 按 F5 启动调试，待 Linux 0.11 完全启动后，执行 mem 应用程序，运行结果如图 8-2所示。请读者理解Linux内核函数get\_free\_page和free\_page的功能和用法。 

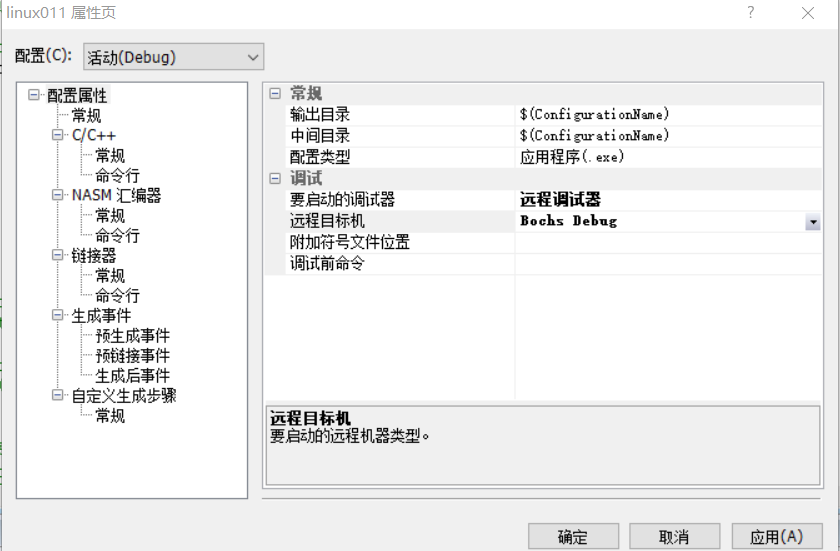
图 8-2 分配一个物理页和释放一个物理页后物理存储器的变化情况

3.3 跟踪逻辑地址、线性地址、物理地址的映射过程 在学习了Linux 0.11管理物理存储器的方式后，接下来重点理解一下从逻辑地址到线性地址，再到物理地址的映射过程。简要过程如下：

首先使用Bochs虚拟机的Debug功能启动Linux 0.11，然后运行一个应用程序，其main函 数中主要包括了一个无限循环，源代码如下： #include <stdio.h> int i = 0x12345678; int main(void) { printf("The logical/virtual address of i is 0x%08x\n", &i); fflush(stdout); while(i) ; return 0; } 应用程序开始运行后，可以使用Bochs调试器让应用程序暂停，然后通过使用Bochs提供 的调试命令，从变量i的逻辑地址计算出它的线性地址，再计算出物理地址。最后，直接修 改物理内存让变量i的值变为0，从而结束死循环使应用程序退出。请读者按照下面的详细步 骤进行实验。

3.3.1 使用 Bochs Debug 作为远程目标机 按照下面的步骤将调试时使用的远程目标机修改为 Bochs Debug：

1. 在 Linux Lab 的“项目管理器”窗口中右键点击项目节点，在快捷菜单中选择“属 性”。

2. 在弹出的“属性页”对话框右侧的属性列表中找到“远程目标机”属性，将此属性 值修改为“Bochs Debug” 。 3. 单击“确定”按钮关闭“属性页”对话框。注意，此时就不能再使用 Linux Lab 提供的可视化调试功能来调试 Linux 0.11 内核了，而只能使用 Bochs 提供的调试 命令进行调试。

3.3.2 启动应用程序并暂停 按照下面的步骤启动应用程序，然后使之暂停：

1. 按 F5 启动调试。此时 Bochs 的显示窗口中无内容，而 Bochs 的命令窗口显示将要 执行的 BIOS 的第一条指令，并等待用户输入调试命令，如下图：



图 8-3 Bochs 的命令窗口等待用户输入调试命令

2. 输入命令“c”按回车，Linux 0.11会继续运行直到等待用户输入命令。

3. 在Linux 0.11中使用vi编辑器新建loop.c文件，并输入之前给出的源代码。

4. 使用GCC将loop.c编译为loop可执行文件并运行，会打印如下信息： 

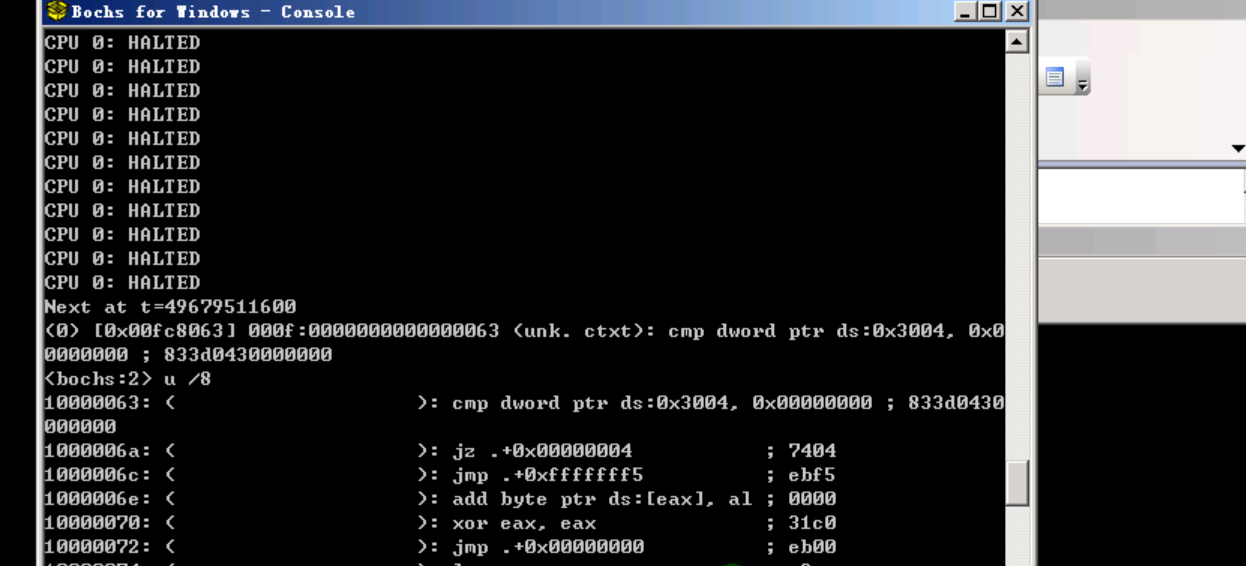
图8-4

loop.c的执行结果 只要loop应用程序不发生改变，0x00003004这个值在任何一个机器上都是一样的。 即使在同一个机器上多次运行，也是一样的。

5. 在Bochs的命令窗口(标题为“Bochs for Windows - Console”)中按“Ctrl+c”， Bochs会暂停运行，进入调试状态。绝大多数情况下都会停在应用程序的死循环内， 显示类似如下信息：

图8-5 Bochs命令窗口在暂停运行后显示的信息 其中的“000f”如果是“0008”，则说明中断在了内核里。那么就要在Bochs的命 令窗口中输入命令“c”后回车，然后再按“Ctrl+c”，直到变为“000f”为止。如 果显示的将要执行的下一条指令不是“cmp …”，就用“n”命令单步运行几步， 直到停在“cmp …”指令处。

6. 使用命令“u/8”，显示从当前位置开始的8条指令的反汇编代码，如下：



Linux 内核实验教程

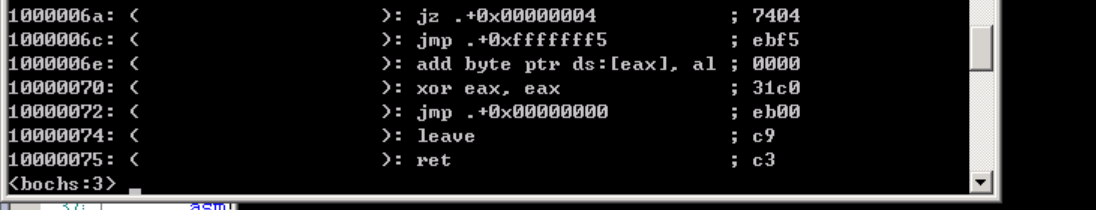


图8-6

从当前位置开始的8条指令的反汇编代码 这就是loop.c中从while开始一直到return的汇编代码。变量i保存在ds:0x3004这 个地址，并不停地和0进行比较，直到它为0，才会跳出循环。

3.3.3 通过段表（GDT和LDT）将逻辑地址映射为线性地址

现在，已经找到了变量i的逻辑地址ds:0x3004。其中，DS寄存器中存储了一个段选择符 （也叫段选择子），可以通过这个段选择符在对应的段描述符表中定位一个段描述符，然后 再从段描述符中获得段基址，最后让段基址与偏移0x3004相加，就可以获得变量i的线性地址了在开始将逻辑地址映射为线性地址前，需要先掌握段选择符和段描述符的格式。段选择符的大小是2个字节（16位），会被装入CS、DS、SS等段寄存器中，所以和这些段 寄存器的大小是一样的。其中,RPL占用最低的2位，表示请求特权级，不参与地址映射的过 程。第2位是TI标志，如果TI为0，表示段描述符在GDT（全局描述符表）中，如果TI为1，表 示段描述符在LDT（局部描述符表）中。第3到15位表示段描述符在GDT或LDT中的索引。

段描述符通用格式 段描述符是一个64位的二进制数据，主要存放了段基址和段限长等信息。其中，段基址占用了32位，由于这32位是分离的，所以需要按照顺序组装起来才能得到完整的段基址；段限 长占用了16位；位P（Present）用来表示段是否存在；位S用来表示段是系统段描述符（S=0） 还是代码或数据段描述符（S=1）；四位 TYPE用来表示段的类型，如数据段、代码段、可读、 可写等；两位DPL是段的权限，和CPL、RPL对应使用；位G是段限长的粒度，G为0表示段限长 以位为单位，G为1表示段限长以4KB为单位；其他内容就不详细解释了。

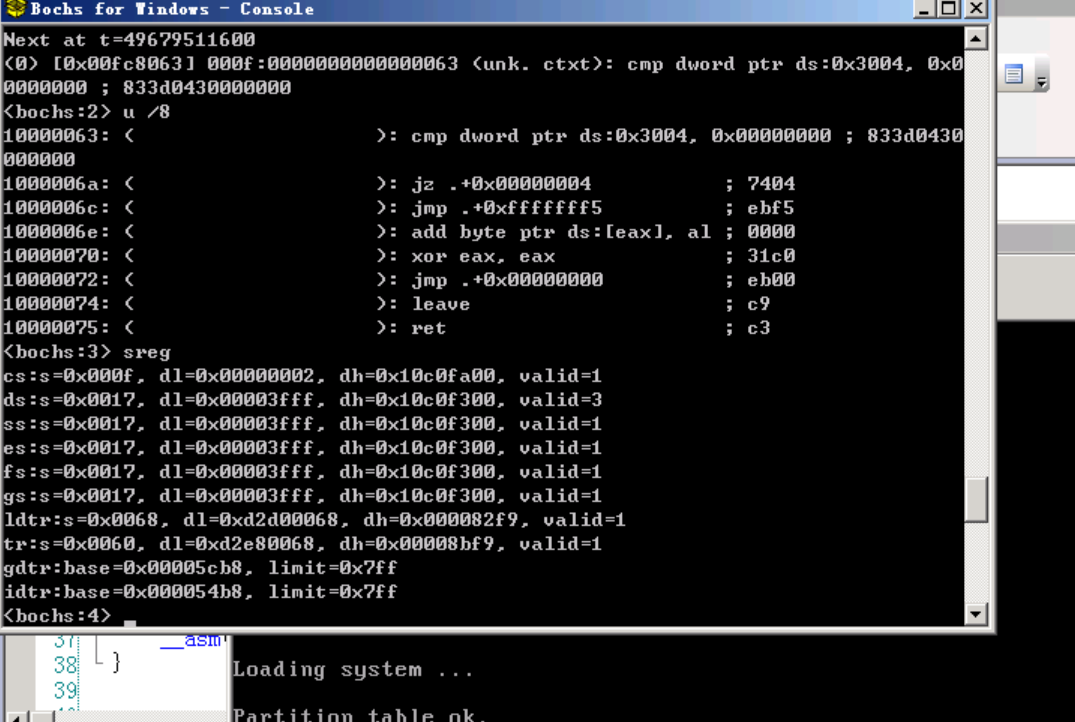
在了解了段选择符和段描述符后，就可以按照下面的步骤操作了： 1. 在Bochs的命令窗口中输入命令“sreg”查看DS寄存器的值，如下图： 

图8-10

段寄存器内的值 DS的值为0x0017，换算为16位的二进制为0000000000010111。其中RPL的值为二进 制11，十进制为3，是最低的特权级（因为在应用程序中执行）；TI的值为1，表示 查找LDT表；使用粗体表示的就是索引值，二进制为10，换算为十进制为2，表示找 LDT表中的第3个段描述符（从0开始编号）。 2. 由于LDT表的基址也是由一个段描述符来描述的，而且这个段描述符存储在GDT表中， 其索引由ldtr寄存器确定。所以，从上图中也可以得到ldtr的值为0x0068，换算为 16位的二级制为0000000001101000，使用粗体表示的就是索引值，二进制为1101， 换算为十进制为13，表示LDT表的描述符在GDT的索引为13（第14个描述符）。 3. GDT的起始物理地址存储在寄存器gdtr中，在上图中显示寄存器gdtr的值为 0x00005cb8。所以在Bochs的命令窗口中输入命令“xp /2w 0x00005cb8+13\*8”， 就可以查看GDT表中索引值为13的段描述符了，如下图：



图8-11

GDT表中索引值为13的段描述符 上面两个命令得到的数值可能和这里给出的示例不一致，这是很正常的。如果读者 需要确认自己得到的值是否正确，可以查看“sreg”命令 输出内容中的ldtr所在行 的dl和dh的值，这两个值是x86处理器为了加快地址映射的速度而存储的段描述符， 读者得到的段描述符的必须和它们一致。

4. 将段描述符“0x82d00068 0x000082fc”中的加粗部分组合为“0x00fc82d0”，就 是LDT表的物理地址了。在 Bochs的命令窗口中输入命令“xp /8w 0x00fc82d0”， 可以得到LDT表中前4个段描述符的内容，如下图：

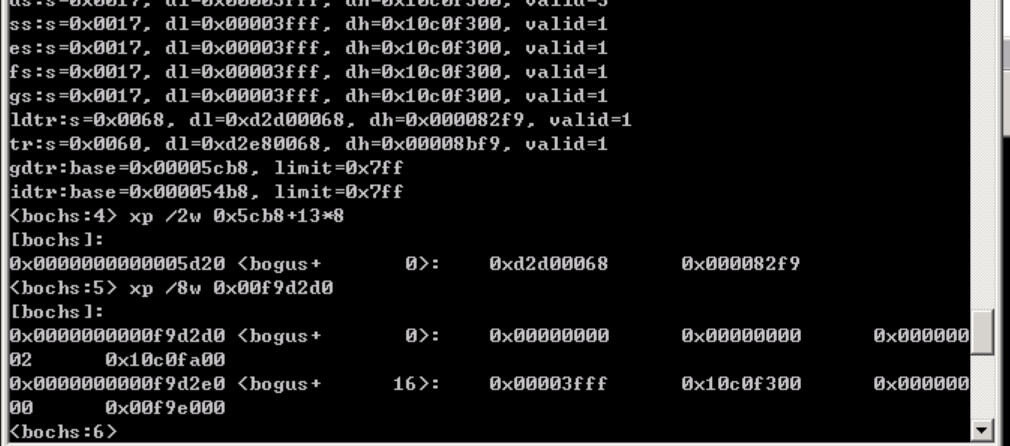
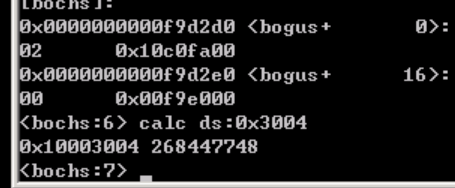


图8-12 LDT表中前4个段描述符 第3个段描述符为“0x00003fff 0x10c0f300”就是ds对应的段描述符了。在Linux 内核实验教程sreg”命令输出的内容中，ds所在行的dl和dh值应该与这个段描述符一致。

段描述符“0x00003fff 0x10c0f300”中加粗部分组合成的“0x10000000”这就是ds段 在线性地址空间中的起始地址。用同样的方法也可以计算其它段的基址，都是这个数。段基 址+段内偏移就是线性地址了。所以ds:0x3004的线性地址就是0x10000000+0x3004 = 0x10003004。在Bochs的命令窗口中输入命令“calc ds:0x3004”可以查看这个逻辑地址的 线性地址，从而可以验证这个结果。



3.3.4 通过页目录和页表将线性地址映射为物理地址 现在，已经得到了变量i的线性地址0x10003004,使用线性地址最高10位的值作为页目录 中的索引（页目录号），就可以得到页表的起始物理地址，再使用线性地址中间10位的值作 为页表中的索引（页表号），就可以找到物理页的起始地址，最后使用线性地址最低12位的 值作为物理页内的偏移（页内偏移），就可以确定最终的物理地址了。如下图所示：

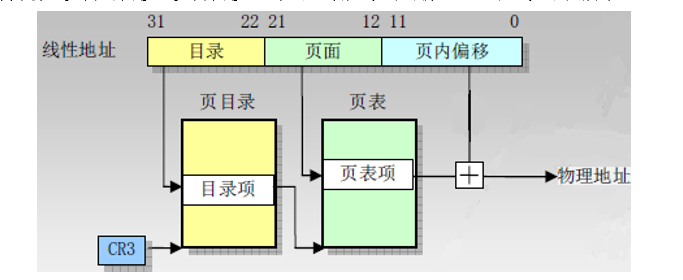


图8-13

线性地址到物理地址的变换过程 在开始将线性地址映射为物理地址前，需要先掌握页目录和页表的格式。 页目录和页表 页目录和页表的格式是一样的，大小均为4KB，包含 1024个页表项，每项有4个字节（32 位）。如下图所示：

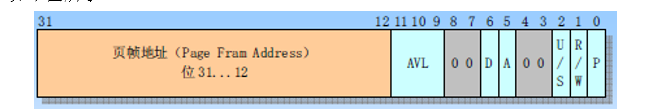


图8-14

页目录和页表中的表项的格式 每个页表项最高的20位是物理页框号，将物理页框号左移12位可以得到其对应物理页的基址， 所以每个物理页的大小也是4KB。页表项中余下的12位是属性位，读者可以自己了解。一个 进程的线性地址空间通常由一个页目录和多个页表来描述，页目录中的页表项用来指定页表 的起始物理地址，页表中的页表项用来指定物理页的起始地址，这样就构成了一个二级页表 映射。页目录的起始物理地址由x86处理器中的控制寄存器CR3指定。

按照下面的步骤将线性地址映射为物理地址： 1. 首先需要算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了32位线 性地址的10位+10位+12位，所以0x10003004的页目录号为64，页表号为3，页内偏 移为4。 2. 页目录表的起始物理地址由CR3寄存器指定。在Bochs的命令窗口中输入命令“creg”

Linux 内核实验教程

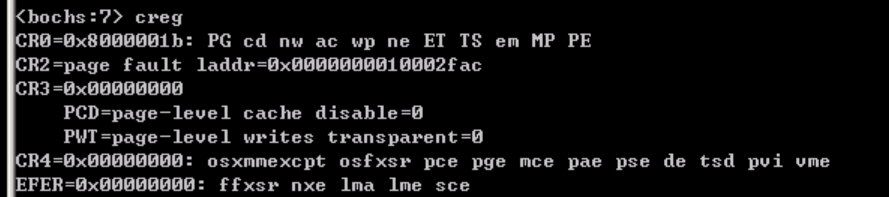
可以查看CR3寄存器的值，如下： 

图8-15

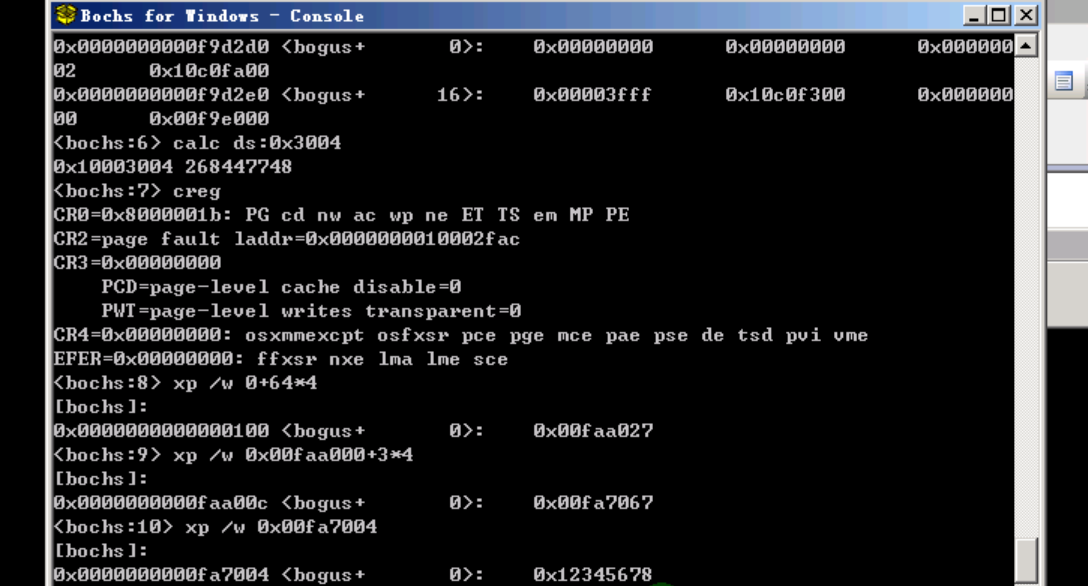
控制寄存器的值 CR3寄存器的值为0，说明页目录的起始物理地址为0。 3. 在Bochs的命令窗口中输入命令“xp /w 0+64\*4”，在页目录中查看页目录号为 64的页表项，如下图（如果读者得到页目录号与这里的不一致，请读者使用自己得 到的值进行后续的计算）： 

图8-16

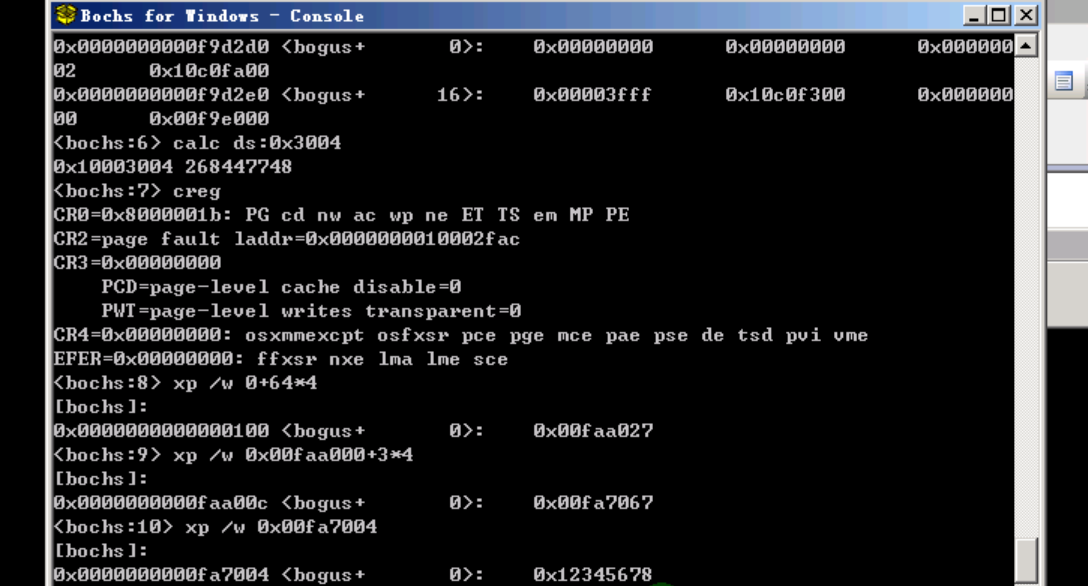
页目录号为64的页表项 其中027是属性位的值，请读者自己分析这些属性值的意义。页表所在物理页框号 为0x00fa7，即页表的起始物理地址为0x00fa7000。 4. 在Bochs的命令窗口中输入命令“xp /w 0x00fa7000+3\*4”，在页表中查看页表号 为3的页表项，如下图： 

图8-17

页表号为3的页表项 其中0x00fa2是物理页所在的页框号，所以物理页的起始地址为0x00fa2000。将物 理页的起始地址与页内偏移4加在一起得到0x00fa2004，这就是变量i的物理地址了。

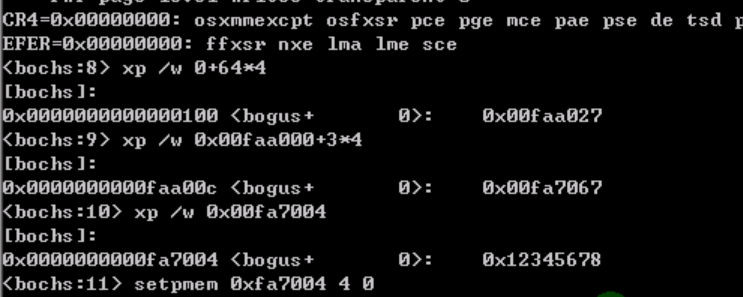
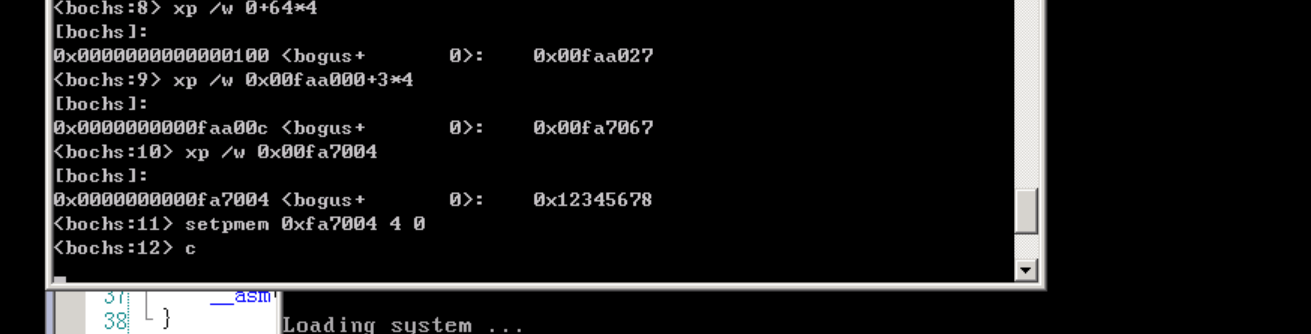
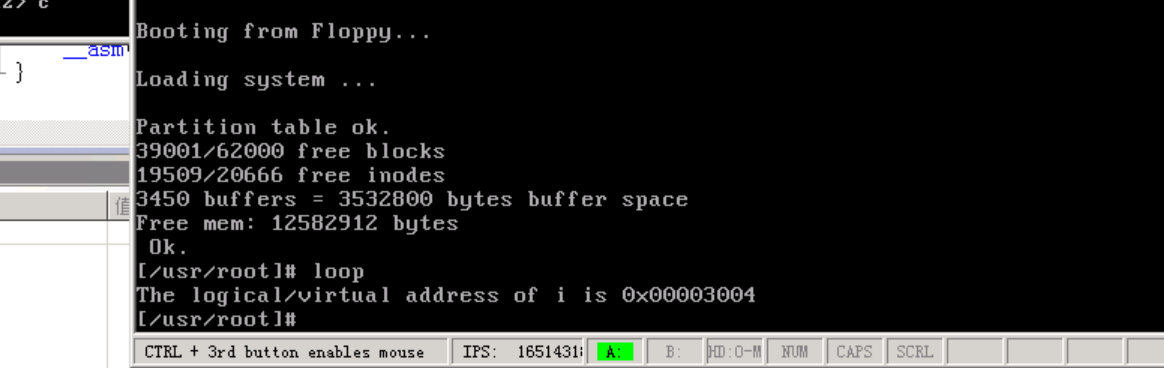
5. 在Bochs的命令窗口中输入命令“xp /w 0x00fa2004”，查看从该物理地址开始 的4个字节的值，也就是变量i的值，如下图： 

图8-18

变量i的值 可以验证这个值与程序中变量i的初值是一样的。 6. 在Bochs的命令窗口中输入命令“setpmem 0x00fa2004 4 0”，将从物理地址 0x00fa2004开始的4个字节的值都设为0。然后再使用命令“c”让Bochs继续运行， 可以看到应用程序退出了，说明变量i的值在被修改为0后结束了死循环。 

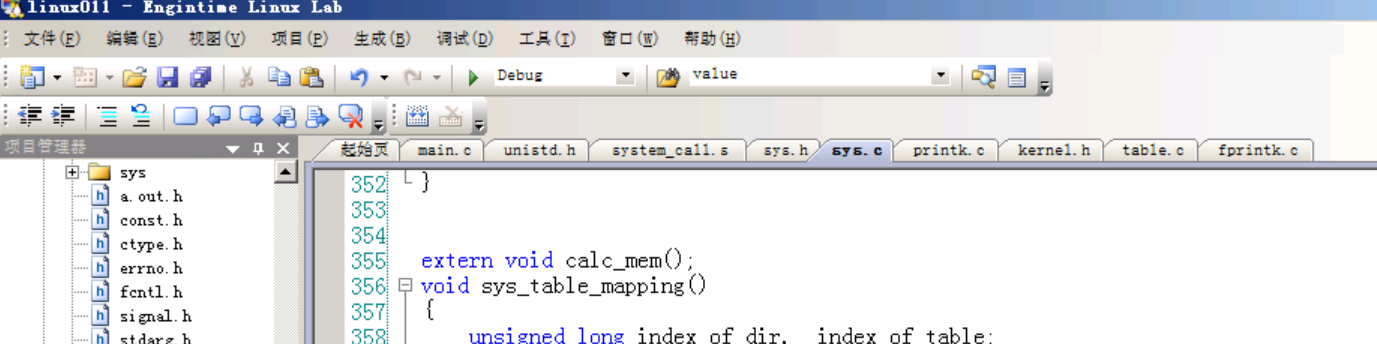
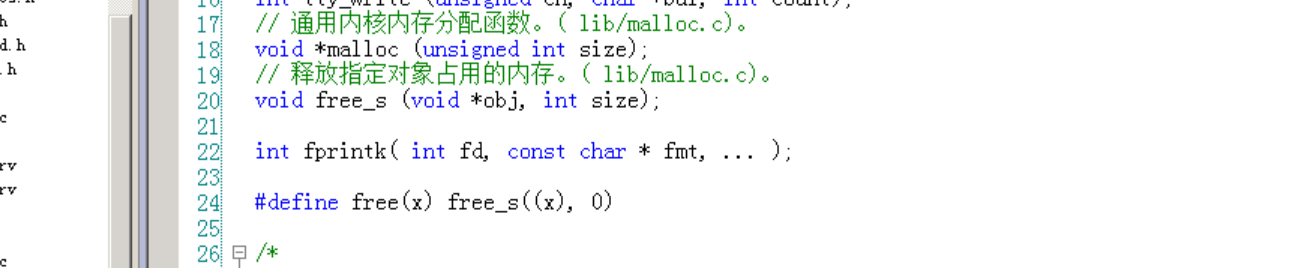
之前调试了全局变量的地址映射过程，下面请读者自己调试局部变量的地址映射过程。 可以将loop应用程序的源代码文件修改为如下的代码，通过物理内存修改局部变量i的值使 应用程序结束。 #include <stdio.h> int main(void) { int i = 0x12345678; printf("The logical/virtual address of i is 0x%08x\n", &i); fflush(stdout); while(i) ; return 0; }

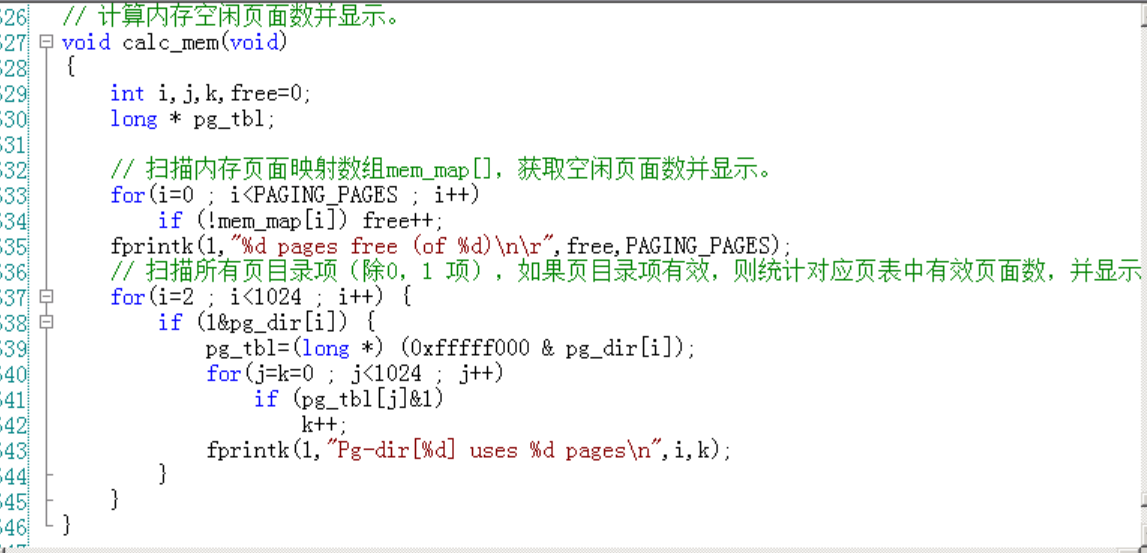
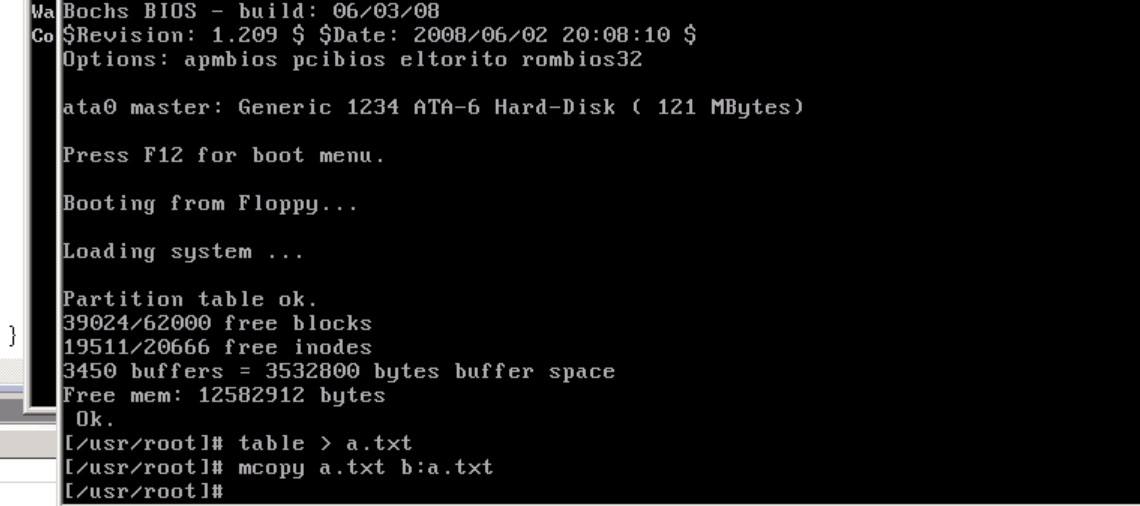


3.4 查看应用程序进程的页目录和页表

之前的练习已经让读者详细了解了逻辑地址、线性地址和物理地址的映射过程。其中逻 辑地址到线性地址的映射过程涉及到了存储器的分段管理，这部分只要知道了段基址和段限 长等概念就可以了。线性地址到物理地址的映射过程涉及到了存储器的二级页表管理，为了 读者能够对二级页表有一个直观的、整体上的认识，并为后面通过共享物理页的方法来实现 内存共享功能，这里使用一个应用程序将进程的二级页表打印出来，供读者仔细研究。请按 照下面的步骤进行实验：

1. 新建一个Linux011 Kernel项目。

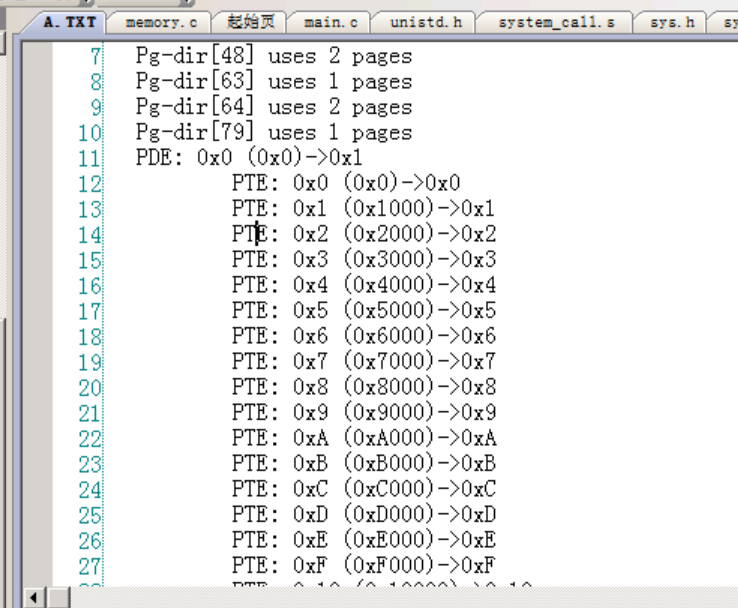
2. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四)，该系统 调用的内核函数sys\_table\_mapping可以写在kernel/sys.c文件的末尾。其源代码可 以参见“学生包”中本实验对应文件夹下的table.c文件。 3. 在sys\_table\_mapping函数中调用了一个名称为fprintk的函数用于向标准输出打印 信息，而没有使用printk函数，这是由于本实验需要输出的内容较多，需要输出到 文件中以便查看，但是printk函数虽然可以在屏幕上进行输出，但是却不能输出到 文件中，所以需要实现一个fprintk函数。该函数可以在kernel/printk.c文件中实 现，其源代码参见“学生包”中本实验对应文件夹下的fprintk.c文件。还需要在 include/linux/kernel.h文件中添加该函数的声明。该函数的第一个参数fd是文件 描述符，其值为1时将输出的信息写入标准输出stdout。  
 4. 在sys\_table\_mapping函数的开始位置还调用了calc\_mem函数，此函数会计算内存中 空闲页面的数量以及各个页表中映射的物理页的数量并显示。在此调用此函数是为 了验证sys\_table\_mapping函数输出的页目录和页表的数量是否正确。由于函数 calc\_mem中使用的是printk函数，所以还无法将输出内容保存到文件中，请读者将 其替换为fprintk函数。

5. 源代码修改完毕后按F7生成项目，确保没有语法错误和警告。 6. 按F5进行调试，待Linux011完全启动后，使用vi编辑器新建一个main.c文件。编辑 main.c文件中的源代码如下： #define \_\_LIBRARY\_\_ #include <unistd.h> #define \_\_NR\_table\_mapping 87 \_syscall0(int, table\_mapping) int main() { table\_mapping(); return 0; }

7. 保存main.c文件后退出vi编辑器，依次执行如下命令： gcc main.c -o table sync table > a.txt mcopy a.txt b:a.txt

8. 结束调试后，在“项目管理器”窗口中双击floppyb.img文件，使用软盘编辑器工具 打开。将其中的a.txt文件复制到Windows的某个文件夹中。

9. 将Windows文件夹中的a.txt文件拖动到Linux lab中释放，分析输出的结果。



在sys\_table\_mapping函数中有一行代码 page\_table\_base = 0xFFFFF000 & entry，是 将一个页表的物理地址转换为它的逻辑地址，其中entry是页目录项，取其中的高20位就可 以得到页表的物理地址。为什么页表的物理地址和它的逻辑地址相同呢？下面结合文件 a.txt中的输出结果进行说明。文件a.txt的开始部分的数据如下：

PDE: 0x0 (0x0)->0x1 PTE: 0x0 (0x0)->0x0 PTE: 0x1 (0x1000)->0x1 PTE: 0x2 (0x2000)->0x2 PTE: 0x3 (0x3000)->0x3 PTE: 0x4 (0x4000)->0x4

一个已知的前提是页目录的物理页框号是0x0，并且页目录的第0项指向物理页框号为0x1的 页表，而此页表的第0项又指回了物理页框号为0x0的页目录，页表的第1项指向物理页框号 为0x1的页表，也就是它自己，并依此类推，即第一个页表中的所有页表项指向了1024个页 表（包括页目录），如图8-19所示。正是由于这样的布局和映射关系，确保了页表的物理地 址与其线性地址相同，而且内核态中段基址为0，所以页表的物理地址也就与其逻辑地址相 同了。例如，逻辑地址0x1000，其线性地址仍然是0x1000，在分页变换时，其高10位为0， 所以在页目录中取偏移为0的页目录项，该页目录项对应的页表的物理页框号为0x1,物理地 址即为0x1000，线性地址中间10位为1，所以在页表中取偏移为1的页表项，该页表项对应的 物理页的页框号为0x1,物理地址仍然为0x1000，可以看到，页表的逻辑地址与物理地址相同。 请读者用逻辑地址0x2000计算一下其对应的物理地址，看看物理页框号为0x2的页表，它的 物理地址与逻辑地址是否相同。

0x0000

0x1000

0x2000

0x3000

……

……

……

. . .

……

页目录

页表

页表

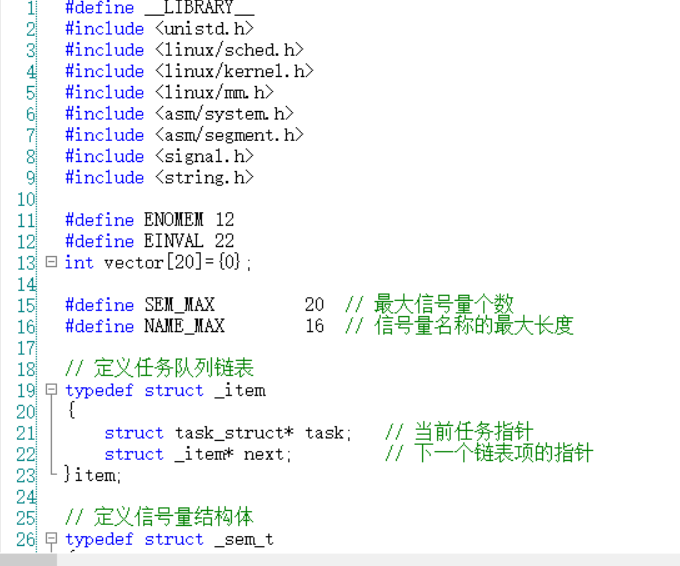
页目录项0 页目录项1

页表项0 页表项1 页表项2

分页后的物理存储器

3.5 用共享内存做缓冲区解决生产者——消费者问题

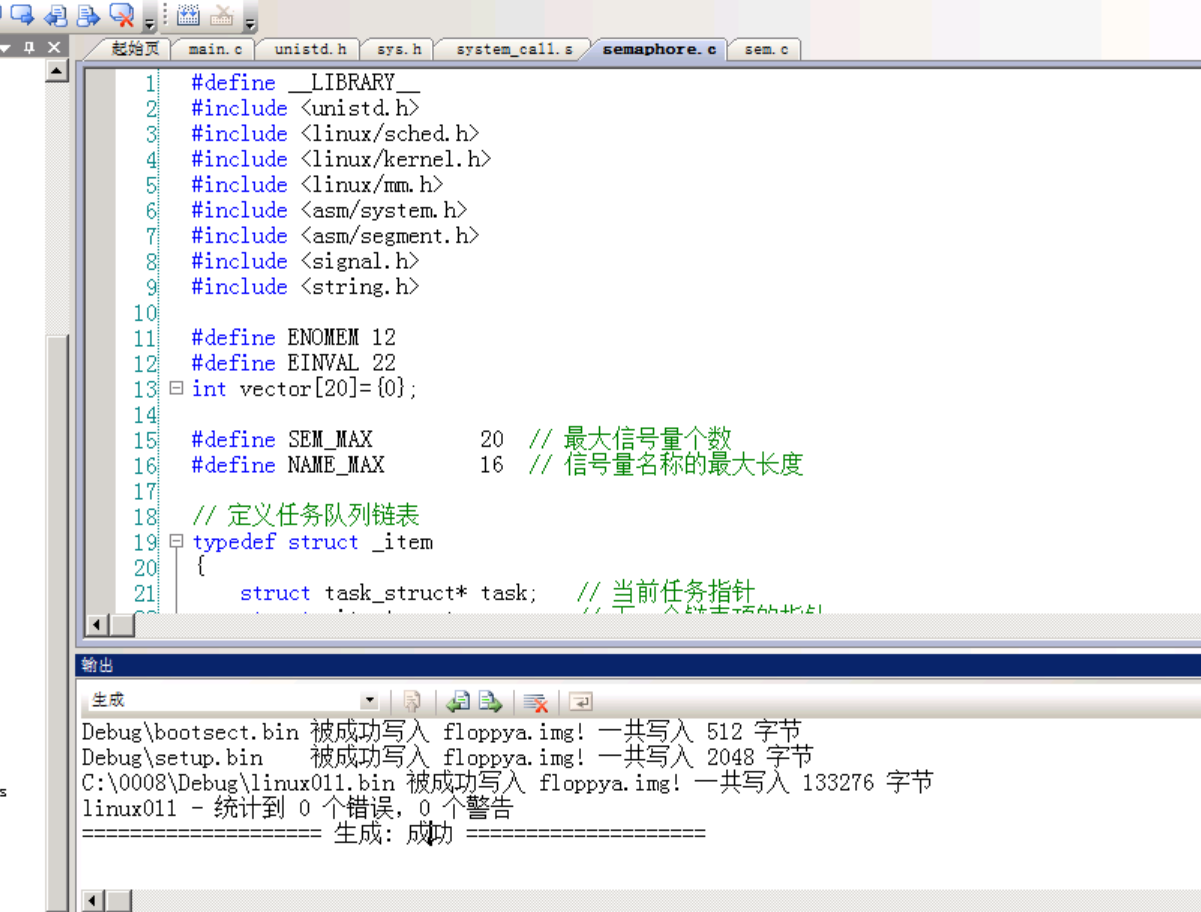
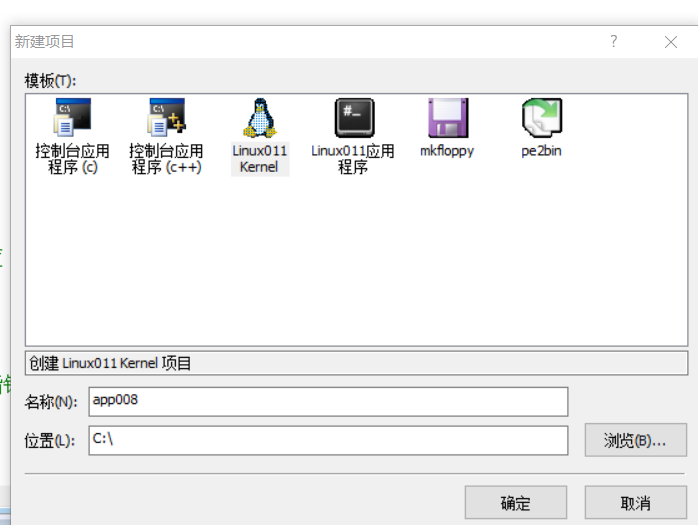
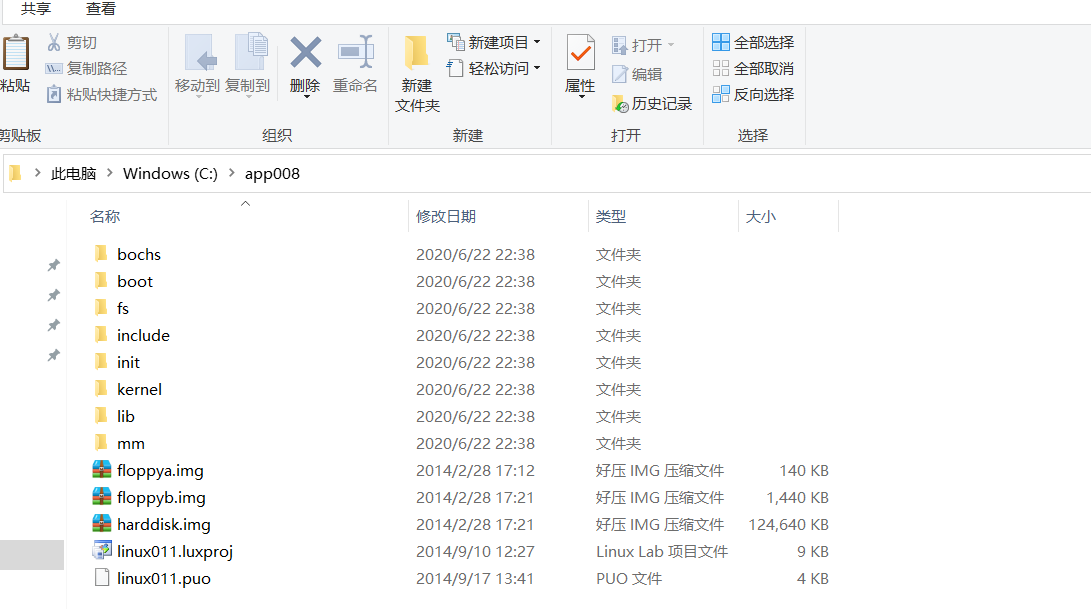
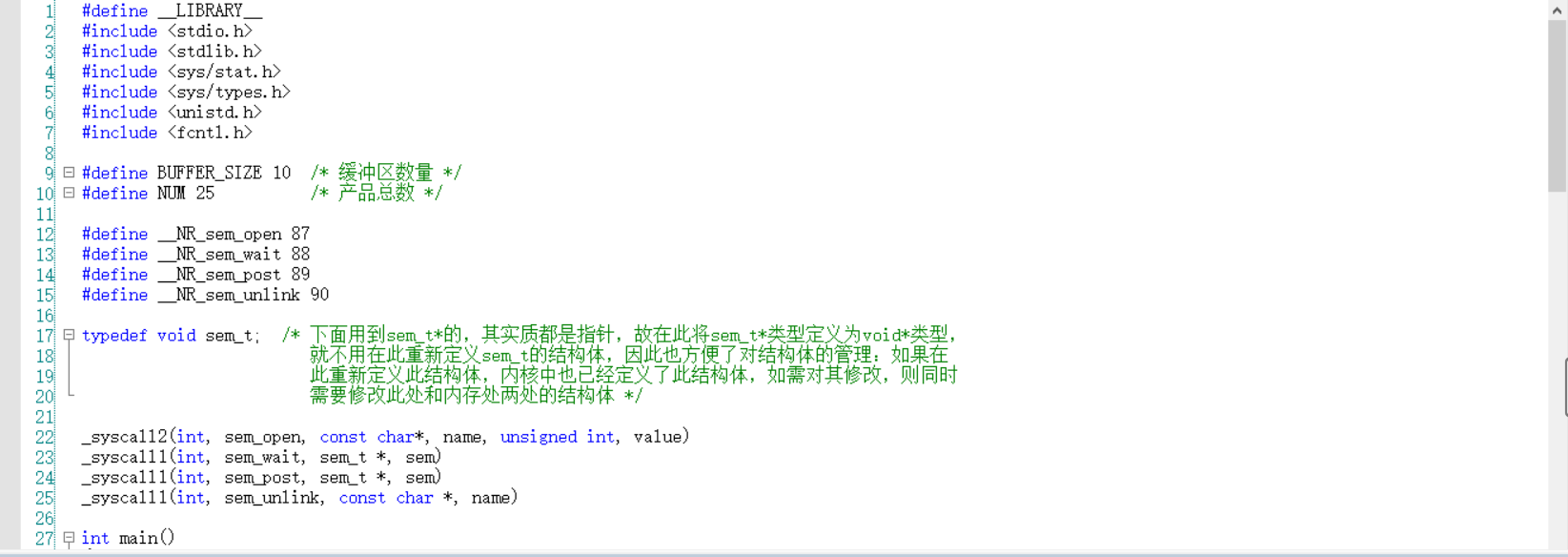
读者通过前面的实验内容学习了Linux 0.11管理物理内存的方式，以及使用x86处理器 提供的分段功能和二级页表功能实现地址映射的过程。在此基础上，读者可以将一个共享的 物理页映射到不同的逻辑地址空间，从而实现在进程间共享内存的方法，进而使用共享内存 作为缓冲区来解决生产者——消费者问题。

本实验需要读者在Linux 0.11的内核中添加shmget与shmat两个共享内存的系统调用函 数。在“学生包”本实验对应的文件夹下提供了一个sem.c文件，其中包含了信号量的四个 系统调用函数（与实验七中的相同），以及共享内存的两个系统调用函数所对应的内核函数 sys\_shmget和sys\_shmat，这里只是实现了一个简化后的版本：只能将一个物理页映射到不 同进程的逻辑地址空间的末尾。请读者仔细阅读其中的源代码，进而理解下面的内容。 系统调用函数shmget的作用是新建或者打开一页共享内存，并将该共享内存的ID返回， 其内核函数sys\_shmget的原型定义如下： int sys\_shmget(int key, int size) 参数key用来指定标识共享内存的的键值，结合源代码可知key就是vector数组的下标，所以 其必须是一个小于数组长度的正整数；参数size用来指定共享内存的大小，由于总是调用 get\_free\_page函数分配一个物理页作为共享内存，所以此参数只是用来进行简单的校验。 在函数的最后将物理页的基地址放入由下标key指定的数组项中，并返回物理地址作为ID。 系统调用函数shmat的作用是将由ID指定的共享内存加入到当前进程的二级页表映射中， 并返回其逻辑地址,其内核函数sys\_shmat的原型定义如下： void\* sys\_shmat(int shmid, const void \*shmaddr) 参数shmid用来指定共享内存的ID，即由 shmget函数返回的ID，其实质是共享页的物理地址； 参数shmaddr在这里没有用到。在这个函数中主要是调用了put\_page函数将物理页加入二级 页表映射。put\_page函数的第一个参数是物理页的基地址，第二个参数是物理页需要映射到 的线性地址。为了简单，这里直接将共享的物理页映射到了应用程序的末尾，所以传入的第 二个参数是current->start\_code + current->brk。其中current->start\_code是应用程序 代码的起始线性地址，current->brk是应用程序所占内存的长度。由于sys\_shmat函数需要 返回共享内存的逻辑地址，而且应用程序在用户态时，其段基址与代码的起始地址相同，即 current->start\_code在应用程序空间的逻辑地址为0，所以 current->brk就是需要返回的逻 辑地址。

请读者按照下面的步骤完成本实验：

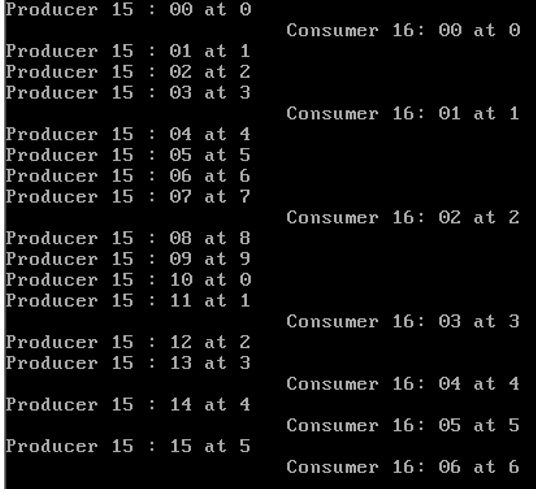
1. 新建一个Linux011 Kernel项目。

2. 将sem.c文件中的四个信号量的系统调用和两个共享内存的系统调用添加到内核中。 添加系统调用的方法可以参考实验四和实验七，具体步骤这里不再详细说明。

3. 按 F7 生成项目，确保没有语法错误和警告。 4. 新建一个 Linux011 应用程序项目，在 windows 资源管理器中将前面 Linux011 Kernel 项目文件夹下 floppya.img 文件拷贝覆盖新建的应用程序项目文件夹下的 floppya.img，这样就可以在应用程序中使用已经添加的信号量和共享内存的系统 调用了。 5. 将“学生包”本实验文件夹下的 pc.c 文件拖放到 Linux Lab 中打开，用其中的源 代码替换刚创建的应用程序项目中的 LinuxApp.c 中的源代码。这些源代码仍然是 使用文件作为生产者和消费者之间的共享缓冲区，请读者在此基础上将其修改为使 用共享内存作为缓冲区。 提示：  在应用程序中定义共享内存系统调用函数的调用号和函数时，可以参考下面的 代码： #define \_\_NR\_shmget 91 #define \_\_NR\_shmat

5.\_syscall2(int, shmget, int, key, int, size) \_syscall2(int, shmat, int, shmid, const void\*, shmaddr)  在生产者进程和消费者进程中都需要分别调用 shmget 函数和 shmat 函数来得 到共享内存的起始地址，可以参考下面的代码： shmid = shmget(key, 4096); startaddress = shmat(shmid, NULL);

6. 代码修改完毕后，按 F7 生成应用程序项目，按 F5 启动调试。

7. 依次执行下面的命令： mcopy b:linuxapp.exe app chmod +x app app 应确保程序执行的结果与实验七中的图 7-1 显示的结果一致。

注意：此应用程序执行到最后可能会出现错误提示“trying to free free page”，这 是由于生产者进程和消费者进程在退出时都会释放共享的物理页，导致一个物理页被回 收两次。为了避免操作系统提示此错误信息并中断执行，可以暂时把文件 mm/memory.c 中 free\_page 函数的最后一行（第 141 行）代码注释掉。

**四、思考与练习**

1. 请读者认真体会本实验中关于地址映射的内容，尝试列出映射过程中最为重要的几步 （不超过四步），并给出获得的实验数据。

2. 在本实验第3.2节的loop程序退出后，如果读者接着再运行一次，并再次进行地址跟踪， 会发现有哪些异同？尝试说明原因？

3. 参考下面的代码，在Linux 0.11内核中写一个系统调用函数，并在Linux 0.11的应用程 序中调用此函数，然后仿照本实验第3.2节的内容跟踪变量i的逻辑地址、线性地址、物 理地址的映射过程，最后通过将物理内存中变量i的值修改为0的方式使应用程序退出。 volatile int i = 0x12345678; void sys\_testg() { printk("The logical/virtual address of i is 0x%08x\n", &i); while(i) ; } 定义变量 i 时使用“volatile”关键字目的是防止编译器对其进行优化，否则 i 的值会 被拷贝到寄存器中，而通过Bochs调试命令修改的是内存中i的值，不是寄存器中的值， 会导致程序无法退出死循环。

4. 将上一个练习中的变量 i 修改为 sys\_testg 函数内的局部变量，仍然通过将物理内存中 变量 i 的值修改为 0 的方式使应用程序退出。请读者结合 Linux 0.11 内核的逻辑地址 空间的内存布局，体会内核中的全局变量与局部变量在逻辑地址空间中的位置有什么不 同。

5. 将本实验3.5中的Linux 0.11应用程序修改为一个生产者与多个消费者使用共享内存作 为缓冲区的情况。（ 提示：关键是需要将多个消费者从缓冲池中取产品的位置放入共享 内存中的适当位置。） 6. 在本实验3.5中遇到了共享的物理页被多个进程重复释放，导致操作系统报告错误并中 断运行的问题，暂时是通过注释掉文件mem/memory.c中free\_page函数的最后一行（第141行）代码来解决的。请读者实现一个简化版本的关闭共享内存的系统调用函数shmdt， 其函数原型可以为： void shmdt(int key, const void\* startaddr) 参数key是共享内存的键值，即为共享内存数组的下标；参数startaddr是共享内存的逻 辑地址。在每个生产者进程和消费者进程退出前，都必须调用此函数关闭共享内存，最 终确保无论是一个生产者与一个消费者同步运行，还是一个生产者与多个消费者同步运 行都不会再出现应用程序退出时操作系统报告错误的情况。

提示：

1. 读者需要实现一个将共享的物理页从当前进程的二级页表映射中移除的内核函数, 其原型可以为： void cancel\_mapping(const void\* linearaddr) 参数linearaddr是需要取消映射的物理页的线性地址，根据此线性地址找到对应的 页表项，然后将页表项置为0即可。

2. 之前的源代码中只是用全局的vector数组保存了共享内存的物理页的基址，现在需 要为共享内存定义一个结构体，其中除了保存物理页的基址外，还需要保存共享内 存的引用计数（可以参考信号量的引用计数），再使用此结构体定义一个全局的共 享内存数组。当进程调用shmdt函数关闭共享内存时，首先将共享内存的引用计数 减1，然后调用cancel\_mapping函数将共享的物理页从二级页表映射中移除。由于 shmdt 的第二个参数startaddr 是共享内存的逻辑地址，需要为其加上 current->start\_code转换为线性地址后，才能作为cancel\_mapping函数的参数。 最后，判断引用计数的值如果大于0，说明仍然有其他进程在使用此共享内存，就结束（不释放物理页）；否则，需要调用free\_page函数释放物理页。

1. **实验总结**

通过做这次实验，我深入理解操作系统的段、页式内存管理。对于段表、页表，以及逻辑地址、线 性地址、物理地址的映射过程我掌握的更加熟练，也更熟悉它们的运用。在实验中我学会了 查看二级页表映射信息，理解页目录和页表的管理方式，通过我本次实验的编程，我实现了段、页式内存管理上的内存共享，从而帮助我更深入理解操作系统的内存管理。